JDK 8 LTS to the latest Performance and Responsiveness Prospective

Anil Kumar
Datacenter Performance
Anil.Kumar@intel.com

Monica Beckwith
JVM Performance
java-performance@Microsoft

@RajputAnilK
@mon_beck

* All trademarks are the property of their respective owners
Pre-production deployment

Run to run variability (Normalized)

- Full capacity metric
- Operational range metric

Worst run
Best run

QCon SFO 2019
What test is running?

Thousands of apps from small data footprint (Heap 2GB) to very large data footprint (Heap 100GB)

Test: a representative *benchmark*!
Deployment environment?

Container

Hypervisor/Guest OS

App

JVM

1 2 3

OS

2nd Generation Intel® Xeon® Scalable Processor

Intel® UPI

2nd Generation Intel® Xeon® Scalable Processor

QCon SFO 2019
Deployment environment?

CPU threads?
Container, Guest OS policies, Pinning

Heap memory?
Guest OS policies, Memory fragmentation, Transparent large pages
Agenda

JDK 8 LTS to latest high level changes

Data using various benchmarks

Explanation for expected and/or strange behaviors

Summary
JDK 8 LTS to the latest ...

Monitoring, code readability and debugging

New usages
Containers   FaaS (Function as a Service)
Microservices Polyglot programming

Performance
Concurrency → Fork/Join → Parallel Streams → Project Loom

Value Types
Networking: Java I/O → NIO → Netty

QCon SFO 2019
Java SE JDK 8 as base for normalization

Currently most use Java SE?

- 8: 79%
- 7: 9%
- 6 or lower: 3%
- 9: 4%
- 10: 4%
- 11 EA: 0%
- We don’t: 1%
Avoid workloads with high variability

Java* JMH* Benchmark

Configuration setting “"-f 1 -wi 3 -w 5s -i 2 -r 15s -t 1"”

<table>
<thead>
<tr>
<th>Class</th>
<th>Speedup</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.openjdk.bench.java.util.stream.AllMatcher.seq_filter_findFirst</td>
<td>1.0x</td>
<td>Run 1</td>
</tr>
<tr>
<td>org.openjdk.bench.java.util.stream.AllMatcher.seq_filter_findFirst</td>
<td>1.5x</td>
<td>Run 2</td>
</tr>
<tr>
<td>org.openjdk.bench.vm.lambda.invoke.Function1.mref_bndLL_IL</td>
<td>1.0x</td>
<td>Run 1</td>
</tr>
<tr>
<td>org.openjdk.bench.vm.lambda.invoke.Function1.mref_bndLL_IL</td>
<td>1.5x</td>
<td>Run 2</td>
</tr>
</tbody>
</table>

* All trademarks are the property of their respective owners
Avoid heap allocation variability

Significant impact can result from variable heap allocation

For larger than 20GB heaps
System in use for long time
Transparent large pages in use

* All trademarks are the property of their respective owners
JDK 8 LTS vs. 11 LTS vs. 12 vs. 13 compare

Performance: Throughput
SPECjvm2008: compute + memory

JDK 11 LTS, 12 and 13 > JDK 8 LTS

6 minutes execution time for each worklet (sufficient optimization time)
Command line option for SPECjvm2008

RUN_OPTS="-showversion"

${JAVA} ${RUN_OPTS} -jar SPECjvm2008.jar -ict -coe \\
startup.helloworld startup.compress startup.crypto.aes startup.crypto.rsa startup.crypto.signverify \\
startup.mpegaudio startup.scimark.fft startup.scimark.lu startup.scimark.monte_carlo startup.scimark.sor \\
startup.scimark.sparse startup.serial startup.sunflow startup.xml.transform startup.xml.validation \\
compress crypto.aes crypto.rsa crypto.signverify derby mpegaudio scimark.fft.large scimark.lu.large \\
scimark.sor.large scimark.sparse.large scimark.fft.small scimark.lu.small scimark.sor.small \\
scimark.sparse.small scimark.monte_carlo serial sunflow xml.transform xml.validation
JDK 8 LTS vs. 11 LTS vs. 12 vs. 13 compare

Performance, Responsiveness, and Variability
SPECjbb*2015: quick summary

Rough estimation of high bound settled performance

critical-jOPS (Responsiveness)

max-jOPS (Full system capacity)

* All trademarks are the property of their respective owners
**SPECjbb2015: JDK 8 LTS \(\rightarrow\) JDK 11 LTS**

*Full system capacity improved up to \(~5\%\)*

*Responsiveness improved up to \(\sim35\%\)*

```
-Xmx150g  -Xms150g  -Xmn130g```

*All trademarks are the property of their respective owners*
Variability: JDK 8 LTS → JDK 11 LTS

SPECjbb2015

<table>
<thead>
<tr>
<th></th>
<th>Full System Capacity</th>
<th>Responsiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

JDK 11 LTS significantly less variability than JDK 8 LTS for responsiveness

% STD Dev

<table>
<thead>
<tr>
<th></th>
<th>Full System Capacity</th>
<th>Responsiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK 8 LTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDK 11 LTS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* All trademarks are the property of their respective owners
DaCapo micro-benchmark: FaaS

Several components execution time as small as 500ms

Startup is similar

Execution time with JDK 11 LTS > JDK 8 LTS (G1GC ?)

* All trademarks are the property of their respective owners
GC Groundwork
Heap Layout

Z GC

Shenandoah GC

G1 GC
G1 Heap Regions

- Eden
- Old
- Humongous
- Survivor
- Eden
- Old

A young generation region
An old generation region
Occupied and Free Regions

- List of free regions
- In case of generational heap (like G1), the occupied regions could be young, old or humongous
GC Commonalities

From

To
Copying Collector aka Compacting Collector aka Evacuation

GC ROOTS

STATIC VARIABLES

THREAD 1 STACK  THREAD N STACK

ANY JNI REFERENCES
Copying Collector aka Compacting Collector aka Evacuation

GC ROOTS

<table>
<thead>
<tr>
<th>STATIC VARIABLES</th>
<th>THREAD 1 STACK</th>
<th>THREAD N STACK</th>
<th>ANY JNI REFERENCES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Copying Collector aka Compacting Collector aka Evacuation
GC Differences

- Marking Algorithms
- GC Phases
- Compaction
- Barriers
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCs</td>
<td>No</td>
<td>Yes</td>
<td>Yes, STW,</td>
<td>Throughput driven</td>
<td>SATB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes, STW,</td>
<td>200ms</td>
<td>Striped</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes, Concurrent,</td>
<td>10ms</td>
<td>Colored Pointers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes, Concurrent,</td>
<td>10ms</td>
<td></td>
</tr>
</tbody>
</table>
Performance!
GC Performance

Throughput and Responsiveness – Higher is Better

```
<table>
<thead>
<tr>
<th></th>
<th>Max Throughput</th>
<th>Throughput under response time constraints</th>
<th>Responsiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>shenandoah</td>
<td>100%</td>
<td>91%</td>
<td>48%</td>
</tr>
<tr>
<td>z</td>
<td>96%</td>
<td>93%</td>
<td>56%</td>
</tr>
<tr>
<td>parallel, base+ng</td>
<td>135%</td>
<td>124%</td>
<td>52%</td>
</tr>
<tr>
<td>parallel, base+xmng1, base+ng+pausg1, base+ng+pause</td>
<td>122%</td>
<td>116%</td>
<td>49%</td>
</tr>
<tr>
<td>base+ng+pauses</td>
<td>122%</td>
<td>119%</td>
<td>47%</td>
</tr>
<tr>
<td>base+ng+pause</td>
<td>119%</td>
<td>116%</td>
<td>48%</td>
</tr>
</tbody>
</table>
```
AOT Groundwork
OpenJDK JIT Compilation (prior to Tiered Compilation)
Tiered Compilation without AOT

Tiered Compilation with AOT

Performance!
AOT Performance

JVM 2008 – Higher is Better

[Bar chart showing performance improvements with AOT and AOT with tiered]
G1 GC and Humongous Objects
What Constitutes a Humongous Object?

- Object < 50% of G1 region size
- Object >= 50% of G1 region size
- Object > G1 region size
What Constitutes a Humongous Region?

- Object $< 50\%$ of G1 region size
- Object $\geq 50\%$ of G1 region size
- Object $> G1$ region size
Humongous Objects

Object NOT Humongous

Object Humongous

Object Humongous -> Needs Contiguous Regions